Module 1: Introduction
Welcome to CS 115

Information about the course is on the website:
https://www.student.cs.uwaterloo.ca/~cs115/

Contact info for all course staff is there, but if in doubt, here are the key email addresses:

   Instructor  Cameron Morland – cjmorland@uwaterloo.ca

   Coordinator  Barbara Daly – bmzister@uwaterloo.ca

   ISAs  cs115@uwaterloo.ca

In this course we will be using DrRacket: racket-lang.org

If you have a laptop, please install DrRacket after class.
Your labs will help you get acquainted with DrRacket.

For today, please immediately go to

   repl.it/languages/scheme
Am I in the right course?

There are alternatives to this course!
CS135 and CS145 cover essentially the same content, but in greater depth:

- Consider CS135 if you’re very comfortable with math (Fall/Winter only)
- Consider CS145 if you love math and want a challenge (Fall only).

If you wish to be a CS major, you do not need to take 135 then CS136; you may be able to take CS115, CS116, then CS136. But taking 135 may be preferable.

If you are not a math student, you may prefer CS105 (Fall/Winter only).
Course components

Lectures  Two 80 minutes lectures per week

Textbook  "How to Design Programs" (HtDP) by Felleisen, Flatt, Findler, Krishnamurthi: www.htdp.org

Class slides  available on the web page and as a printed course pack from MC2018

Labs  80 minutes using DrRacket v6.2 or higher: www.racket-lang.org
Marking Scheme

Details are on the course website.

- 20% Assignments (roughly weekly)
- 30% Midterm
- 5% Participation
- 45% Final
- 3% Lab Bonus

You must pass the weighted exam component to pass the course.

Grade Appeals: Review policy on course web page.
Academic Offenses: Review policy on course web page.
Participation

Your participation mark comes from *clicker* questions in class. The purpose of these questions is encourage participation and provide feedback on your learning.

You receive 2 marks for each correct answer, 1 mark for each incorrect answer. The best 75% of all questions is used to calculate your grade.

You must attend your own section to receive these grades.

Never use another student’s clicker.

Register your clicker on the CS115 website, *not at the central UW website!*
First Week Tasks

Be sure you do all the following:

1. Install DrRacket on your laptop, if you have one: [www.racket-lang.org](http://www.racket-lang.org)
2. Find out about your labs, and participate in the first one.
3. From the website [www.student.cs.uwaterloo.ca/~cs115/](http://www.student.cs.uwaterloo.ca/~cs115/) download the course notes (or buy them from MC 2018) and review the course details, including survival guide, marking scheme, and grade appeals policy.
4. Bookmark the course textbook, [www.htdp.org](http://www.htdp.org), and read the appropriate sections.
5. Register your iClicker, and see how clickers affect your grade. Details are on the course website.
6. Complete Assignment 00.
Most of your learning comes from struggling with material. You learn little from merely copying work from another.

All assignments are to be done individually.

Don’t look at someone else’s programs written for an assignment, or show your programs to someone else. Don’t search on the web or in books other than the textbook for answers to assignment questions, or even for hints.

You must do your own work in this course. Read the section on plagiarism in the CS 115 Survival Guide.

Start your assignments early. Make sure you have time to fix your code if you don’t pass the basic tests! Bring questions to office hours as soon as possible.
Go over your graded assignments and midterm: learn from your mistakes.
Use labs to get practice.
Computers run only **machine code**.

This is machine code, and a representation of it in assembly language:

```
6a0:  55                      push  %rbp
6a1:  48 89 e5              mov  %rsp,%rbp
6a4:  48 83 ec 10           sub  $0x10,%rsp
6a8:  89 7d fc              mov  %edi,-0x4(%rbp)
6ab:  89 75 f8              mov  %esi,-0x8(%rbp)
6ae:  89 55 f4              mov  %edx,-0xc(%rbp)
6b1:  83 7f fc 00           cmpl  $0x0,-0xc(%rbp)
6b5:  75 05                  jne  6bc <fibonacci+0x1c>
6b7:  8b 45 f8              mov  -0x8(%rbp),%eax
6ba:  eb 1a                 jmp  6d6 <fibonacci+0x36>
6bc:  8b 45 f4              mov  -0xc(%rbp),%eax
6bf:  8d 50 ff              lea  -0x1(%rax),%edx
6c2:  8b 4d fc              mov  -0x4(%rbp),%ecx
6c5:  8b 45 f8              mov  -0x8(%rbp),%eax
6c8:  01 c1                add  %eax,%ecx
6ca:  8b 45 f8              mov  -0x8(%rbp),%eax
```
It is easier to write in a higher level language, like C, which is compiled to machine code:

```c
// fibonacci(previous, latest, count): returns the count-th number following latest in the Fibonacci sequence, where previous is the one before latest.
// fibonacci: Nat Nat Nat -> Nat
// Examples:
// fibonacci(0, 1, 0) => 1
// fibonacci(0, 1, 4) => 5
// fibonacci(5, 8, 1) => 13
int fibonacci(int previous, int latest, int count) {
    if (count == 0) {
        return latest;
    } else {
        return fibonacci(latest, previous + latest, count-1);
    }
}
```
The Design Process

We will cover the whole process of designing programs.

clever, ad-hoc solutions to problems (hard to generalize) → design: the intelligent, extendable use of technique

← technique: fixed ways of doing things (little thought required)

Careful use of design processes can save time and reduce frustration, even with the fairly small programs written in this course.
Themes of the course

**Design**  the art of creation
**Abstraction**  finding commonality, ignore irrelevant details
**Refinement**  revising and improving initial ideas
**Syntax**  how to say things
**Expressiveness**  how easy it is to say and understand
**Semantics**  the meaning of what is said
**Communication**  understanding other's programs, and making your programs understandable
Simple Racket programs

We will start by doing some simple calculations. If you have already installed DrRacket, open it now. Otherwise, install it before next class. For today, immediately go to

repl.it/languages/scheme
We will start by doing some simple calculations.
For today, please immediately go to

repl.it/languages/scheme

Vocabulary:

- In $g(x, y) = x - y$, $x$ and $y$ are called the **parameters** of $g$. In a **function application** such as $g(5, 3)$, 5 and 3 are the **arguments** for the parameters.
- We **evaluate** an expression such as $g(3, 1)$ by **substitution**. Thus $g(3, 1) = 3 - 1 = 2$.
- The function **consumes** 3 and 1, and **returns** 2.
There are many built in functions, too many to list here. Basic arithmetic functions include +, -, *, and /.

(+ 3 4) => 7
(- 7 4) => 3
(* 6 7) => 42
(/ 42 7) => 6

In the menus: Help→Racket Documentation.

A few of particular interest:

- (quotient n m) performs integer division (discarding the remainder). For example,
  (quotient 75 7) => 10

- (remainder n m) computes the remainder of integer division. For example,
  (remainder 75 7) => 5
What is wrong with each of the following?

- (* (5) 3)
- (+ (* 2 4))
- (5 * 14)
- (* + 3 5 2)
- (/ 25 0)
Syntax and Semantics

Syntax refers to the ways we may express things. Racket syntax is very simple. Every expression is either a value such as a number, or of the form \((\text{fname} \ A \ B\ldots)\), where \(\text{fname}\) is the name of a function, and \(A\) and \(B\) are expressions. (More later.)

- \((\ast \ (5) \ 3)\) contains a syntax error since 5 is not the name of a function.
- \((5 \ * \ 14)\) has the same syntax error.
- \((+ \ (* \ 2 \ 4))\) contains a syntax error since the brackets don’t match.

Semantics refers to the meaning of what we say. Semantic errors occur when an expression (which has correct syntax) cannot be reduced to a value by substitution rules.

- \((\ast \ + \ 3 \ 5 \ 2)\) contains a semantic error since we cannot multiply “plus” with 3, 5, and 2.
- \((/ \ 25 \ 0)\) contains a semantic error since we cannot divide by zero.
Defining functions in Mathematics

A few important observations:

- Changing names of parameters does not change what the function does. 
  \( f(x) = x^2 \) and \( f(z) = z^2 \) have the same behaviour.
- Different functions may use the same parameter name.
- Parameter order matters. \( g(3, 1) = 3 - 1 \) but \( g(1, 3) = 1 - 3 \).
- Calling a function creates a new value.
Defining functions in Racket

To translate \( f(x) = x^2 \) and \( g(x, y) = x - y \) into Racket:

\[
\text{(define (f x) (* x x ))}
\]
\[
\text{(define (g x y) (- x y))}
\]

`define` is a special form. It looks like a Racket function but not all its arguments are evaluated. It binds a name to an expression. This expression may use the parameters which follow the name, along with other built-in and user-defined functions.

Exercise

Use `define` to create a function `(add-twice a b)` that returns \( a + 2b \).

(add-twice 3 5) => 13
Identifiers

Functions and parameters are named by identifiers, like $f$, x-ray, wHaTeVeR.

- Identifiers can contain letters, numbers, -, _, ., ?, =, and some other characters.
- Identifiers cannot contain space, brackets of any kind, or quotation marks like `"`.

A few examples:

\begin{verbatim}
(define (g x y) (- x y))
(define (square-it! it!) (* it! it!))
(define (2remainder? n) (remainder n 2))
\end{verbatim}
As with Mathematical functions:

- Changing names of parameters does not change what the function does. (define (f x) (* x x)) and (define (f z) (* z z)) have the same behaviour.

- Different functions may use the same parameter name; there is no problem with (define (f x) (* x x))
  (define (g x y) (- x y))

- Parameter order matters. The following two functions are not the same:
  (define (g x y) (- x y))
  (define (g y x) (- x y))
Racket also allows a special kind of value called a **constant**:

```
(define k 3)
```

This binds the identifier `k` to the value `3`.

```
(define p (* k k))
```

The expression `(* k k)` is evaluated, giving `9`. The identifier `p` is then bound to this value.

There are a few built-in constants, including `pi` and `e`. Some programs might make their own constants, such as `interest-rate` or `step-size`.

Constants can make your code easier to understand and easier to change.

Note: what the CS 115 course notes call “constants”, the textbook calls “variables”.
"Big red trucks drive quickly" is an English sentence with correct syntax and clear semantic interpretation.

"colorless green ideas sleep furiously"\(^1\) has the same syntax, but no clear semantic interpretation.

"Students hate annoying professors" and "I saw her duck" both have ambiguous semantic interpretation; they have multiple possible meanings.

Computer languages are designed so every program has at most one semantic interpretation.

\(^1\)from *Syntactic Structures* by Noam Chomsky, 1957.
Exercise

Given these definitions:

(define foo 4)
(define (bar a b) (+ a a b))

What is the value of this expression?

(* foo (bar 5 (/ 8 foo)))
A model of Racket semantics

We wish to be able to predict the behaviour of any Racket program. We can do this by viewing running a program as applying a set of substitution rules. Any expression which is not a definition we will simplify to a single value.

For example, consider

\[(\ast \ 3 \ (+ \ 1 \ (+ \ (/ \ 6 \ 2) \ 5)))\]

Since the semantic interpretation of \( (/ \ 6 \ 2)\) is 3, we can simplify:

\[\Rightarrow \ (\ast \ 3 \ (+ \ 1 \ (+ \ 3 \ 5)))\]

Ex. Complete the interpretation of \((\ast \ 3 \ (+ \ 1 \ (+ \ (/ \ 6 \ 2) \ 5)))\)
Now consider the following program:

\[
\begin{align*}
&\textbf{define} \ (f \ x) \ (\ast \ x \ x) \\
&\textbf{define} \ (g \ x \ y) \ (- \ x \ y) \\
&g \ (f \ 2) \ (g \ 3 \ 1)
\end{align*}
\]

The function \((f \ 2)\) is bound to \((\ast \ 2 \ 2)\), so simplify:

\[
\Rightarrow \ (g \ (\ast \ 2 \ 2) \ (g \ 3 \ 1))
\]

Ex. Complete the interpretation of \((g \ (f \ 2) \ (g \ 3 \ 1))\)
Rules for evaluating a Racket expression

Goal: a unique sequence of substitution steps for any expression.
Recall from before:
“Every expression is either a value such as a number, or of the form \( (\text{fname} \ A \ B...) \), where \( \text{fname} \) is the name of a function, and \( A \) and \( B \) are expressions.”

Major approach: to evaluate an expression such as \( (\text{fname} \ A \ B) \)

1. evaluate the arguments \( A \) and \( B \), then
2. apply the function to the resulting values.

For example, to evaluate \( (+ \ (/ \ 6 \ 2) \ 5) \), first we need to evaluate \( (/ \ 6 \ 2) \), which gives 3. The other argument, 5, is already evaluated. The expression becomes \( (+ \ 3 \ 5) \), so apply the + function to the values 3 and 5, giving 8.

Note: we do not evaluate definitions; we use definitions to evaluate expressions.
Let’s all do it the same way

Evaluate arguments starting at the left.

For example, given (* (+ 2 3) (+ 5 7)), perform the substitution (+ 2 3) ⇒ 5 before the substitution (+ 5 7) ⇒ 12.

(Note: for functions, this choice is arbitrary, and every choice will give the same final value. But if we all do it the same way it’s easier to communicate what we are doing. Some special forms, discussed later, must be evaluated left-to-right.)
Substitution rules

Built-in function application use mathematical rules.

E.g. \((+\ 3\ 5) = 8\)

Value no substitution needed.

Constant replace the identifier by the value to which it is bound.

\((\text{define } x\ 3)\)
\((*\ x\ (+\ x\ 5))\)

\(\Rightarrow\ (*\ 3\ (+\ x\ 5))\)
\(\Rightarrow\ (*\ 3\ (+\ 3\ 5))\)
Substitution rules

**User-defined function application**  a function is defined by \((\text{define} \ (f \ x1 \ x2 \ldots \ xn) \ \text{exp})\).

Simplify a function application \((f \ v1 \ v2 \ldots \ vn)\) by replacing all occurrences of the parameter \(x_i\) by the value \(v_i\) in the expression \(\text{exp}\). For example,

\[
(\text{define} \ (\text{foo} \ a \ b) \ (+ \ a \ (- \ a \ b)))
\]

\[
(\text{foo} \ 4 \ 3)
\]

\[
⇒ \ (+ \ 4 \ (- \ 4 \ 3))
\]

Note: each \(v_i\) must be a value. To evaluate \((\text{foo} \ (+ \ 2 \ 2) \ 3)\), *do not* substitute \((+ \ 2 \ 2)\) for \(a\), to give \((+ \ (+ \ 2 \ 2) \ (- \ (+ \ 2 \ 2) \ 3))\).

Always evaluate the arguments first.

\[
(\text{foo} \ (+ \ 2 \ 2) \ 3)
\]

\[
⇒ \ (\text{foo} \ 4 \ 3)
\]

\[
⇒ \ (+ \ 4 \ (- \ 4 \ 3))
\]

\[
⇒ \ (+ \ 4 \ 1)
\]

\[
⇒ \ 5
\]
Beware of tricksters...

Type in the code and see what each evaluates to.

Exercise

```
(define x 4)
(define (f x) (* x x))
(f 3)
```

Exercise

```
(define (huh? huh?) (+ huh? 2))
(huh? 4)
```

Exercise

```
(define y 3)
(define (g x) (+ x y))
(g 5)
```

Exercise

```
(define z 3)
(define (h z) (+ z z))
(h 7)
```
Applying simplification rules such as these allows us to predict what a program will do. This is called **tracing**.

Tracing allows you to determine if your code is semantically correct – that it does what is supposed to do.

If no rules can be applied but an expression cannot be further simplified, there is an error in the program. For example, `(sqr 2 3)` cannot be simplified since `sqr` has only one parameter.

Be prepared to demonstrate tracing on exams.

Racket has a feature call the *Stepper* that traces code automatically. This does not always use the same tracing rules as described here, even if it gives the same result. Write your own trace to ensure you understand your code.
Trace the program:

```
(+ (remainder (- 10 2) (quotient 10 3)) (* 2 3))
```
Exercise

Write a Racket function corresponding to

\[ g(x, y) = x \sqrt{x} + y^2 \]

((sqrt n) computes \( \sqrt{n} \)) and ((sqr n) computes \( n^2 \)).
Trace the program: \((\textit{sqrt} \ n)\) computes \(\sqrt{n}\) and \((\textit{sqr} \ n)\) computes \(n^2\)

\[(\textit{define} \ (\textit{disc} \ a \ b \ c) \ (\textit{sqrt} \ (- \ (\textit{sqr} \ b) \ (* \ 4 \ (* \ a \ c))))))\]

\[(\textit{define} \ (\textit{proot} \ a \ b \ c) \ (/ \ (+ \ (- \ 0 \ b) \ (\textit{disc} \ a \ b \ c)) \ (* \ 2 \ a)))\]

\((\textit{proot} \ 1 \ 3 \ 2)\)
Module summary

- You should be able to define and use constants and simple arithmetic functions.
- Become comfortable identifying syntax errors, and expressions which are syntactically correct. Understand the syntax rules we have defined.
- Start getting used to error messages from Racket.
- Be able to trace the substitutions of a Racket program.

Before we begin the next module, please

- Read *How to Design Programs*, Sections 1, 2, 3.
- Read the Survival Guide on assignment style and submission
- Read the Style Guide, Sections 1-4 and 6.